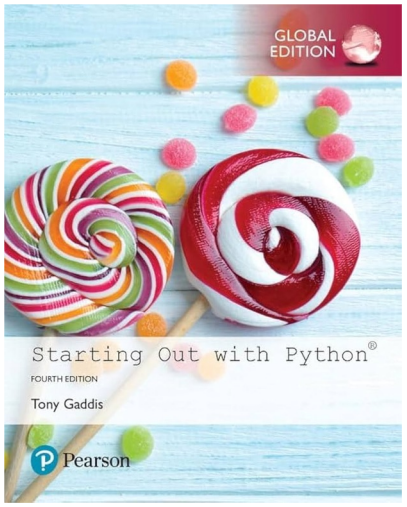
Objects and Classes

CIIC 3015 - Lecture 9

Jean Carlos Méndez

Lecture Goals

- **Classes**
  - What are they?
  - How can they be used?
  - Examples
- **Attributes and Methods**
  - What are they?
  - How can they be used?
  - Examples
- **Inheritance**
  - What is it?
  - How can it be used?
  - Examples



Starting out with Python
(4th Edition)
Chapters 10 and 11

Google | GH |

Quick Recap

Classes

What we know so far...

A whole number? `integer`

A rational number? `float`

A series of characters? `string`

Binary True or False? `boolean`

Ordered collection of items? `list`

Mapping from arbitrary keys to values? `dictionary`

Unordered collection of items without duplicates? `set`

Scenarios

These types have served us well for many common cases.

- Names (string)
- Money (float)
- Counts (int)
- Class Roster (list)

But what if we want to represent something more complicated, like a bank account or a character in a role-playing game (RPG)?

Game Character

Let’s think about designing an RPG.

What properties do characters have?

- Health
- Attack Strength
- Defense

Classes

In our game, all characters have a name, strength level, defense level, and start out with 100 health points.

```
woodie = Character("Woodie", 15, 7)
print(woodie.strength)  # prints 15
print(woodie.health)    # prints 100
print(woodie.name)      # prints "Woodie"
print(woodie.defense)   # prints 7

woodie.introduce()
# prints "Hi, my name is Woodie!"
```

I will call him...Woodie the Lumberjack

Classes

A *class* lets you define your own type.

```
class <Name>:
  <suite>
```

Think of a class to be a bundle of

- class attributes (which hold data)
- class methods (which are functions tied to the class)

The **class definition** serves as a template for its **instances**. Then an **object** is an **instance** of a particular **class**.

Classes

The class definition almost always has a special class method called `__init__`

```
class <Name>:
  def __init__(self, param1, ...):
    self.attribute = param1
    ...
```

`__init__` is a special class method that is called when you create an instance of the class. It is called the **constructor** of the class, and always has `self` as the first parameter (for which you don't pass anything in), followed by any parameters you need to use to *construct* an instance of the class.

Defining a Class

For our character example…

```
class Character:
    def __init__(self, char_name, str_score, def_score):
        self.health = 100
        self.name = char_name
        self.strength = str_score
        self.defense = def_score

    def introduce(self):
        print("Hi, my name is " + self.name)
```

Creating a Class Instance

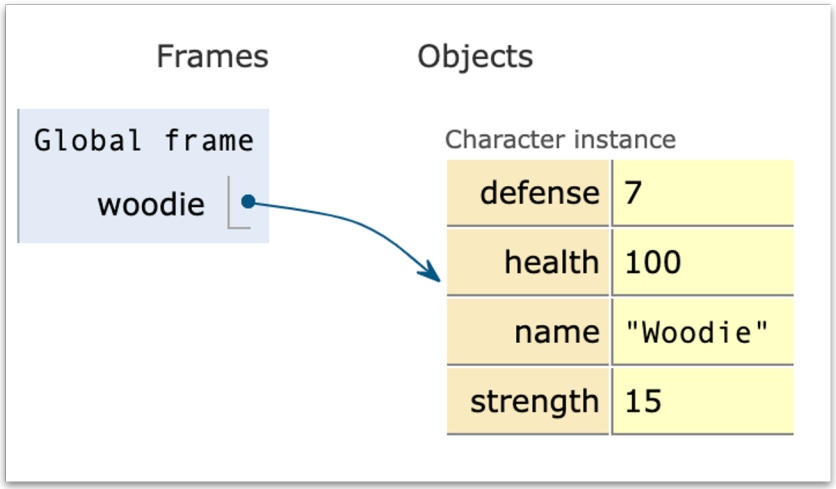
We create an instance of our class by calling the name of the class like a function call, and passing in any arguments that the `__init__` method needs.

```
woodie = Character("Woodie", 15, 7)
pickie = Character("Pickie", 12, 8)
```

PythonTutor Class Visualization

When you create an instance of a class, the **constructor** (aka `__init__` method) is called and executed.

```
woodie = Character("Woodie", 15, 7)
```



Style

For your code to be runnable, a class name must follow the same rules as a variable. However, just like for variables, there are *style conventions* that you should follow to make your code more readable.

Naming convention for classes:

- class names are always upper Camel Case, meaning all words are concatenated and start with a capital letter.
  - “team roster” -> `TeamRoster`
  - “animal” -> `Animal`
  - “movie metadata” -> `MovieMetadata`
  - “very long class name” -> `VeryLongClassName`

Live Coding! Example 1

- Create Character class and print values.

Code available in:

github.com/CSwithJC/CIIC3015/blob/main/lecture_08/example_01.py

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

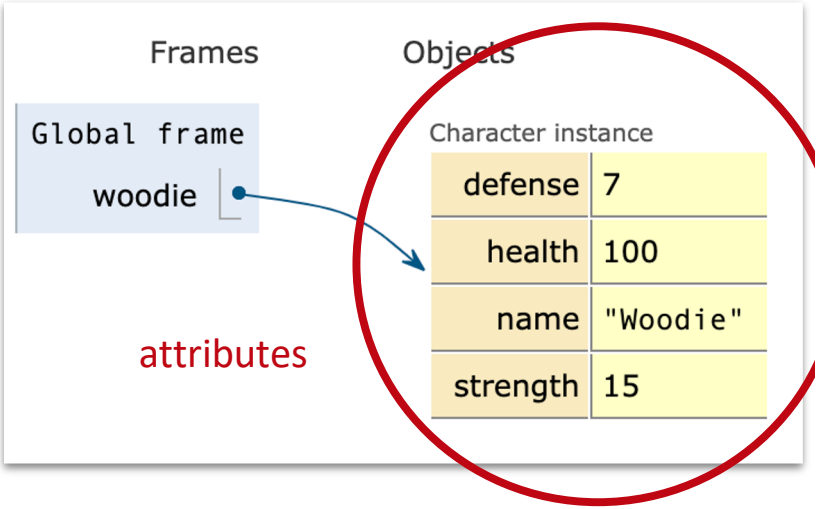
Google | GH |

Class Attributes

Attributes

Attributes don't get stored in any frame. Rather, they are stored as part of the object (similar to a list or dictionary).

```
woodie = Character("Woodie", 15, 7)
```



Two main ways to set attributes:

1. instance attributes via instance object
2. "default" class attributes in class definition

Instance Attributes

These are attributes set by assigning to the instance of a class.

They can be set via:

- `self.attribute_name = value` (inside of a class)
- `variable_name.attribute_name = value` (outside of the class)

They can be accessed via:

`self.attribute_name` (inside of a class)

`variable_name.attribute_name` (outside of a class)

Instance attributes are **unique** to that instance of a class

Instance Attributes

```
class Character:
    birthplace = "Kingdom of Asgardia"
    def __init__(self, char_name, str_score, def_score):
        self.health = 100
        self.name = char_name
        self.strength = str_score
        self.defense = def_score

woodie = Character("Woodie", 15, 7)
woodie.favorite_color = "Blue"
print(woodie.favorite_color) # prints “Blue”
print(woodie.health) # prints 100
woodie.health = 90
print(woodie.health) # prints 90
```

Google | GH |

Class Attributes

These are the kind of attributes set directly in the class definition (not in the constructor and not on the self parameter). Thes can be set via:

- `attribute_name = value` (inside of the class)
- `ClassName.attribute_name = new_value` (either inside or outside the class)

The can be accessed via:

- `ClassName.attribute_name = new_value` wherever
- `variable_name.attribute_name` if there is no instance variable already using that attribute name

Is **shared** among all instances of the class

Class Attributes

```
class Character:
    birthplace = "Kingdom of Asgardia"
    def __init__(self, char_name, str_score, def_score):
        self.health = 100
        self.name = char_name
        self.strength = str_score
        self.defense = def_score

woodie = Character("Woodie", 15, 7)
pickie = Character("Pickie", 23, 4)
print(woodie.birthplace)  # prints “Kingdom of Asgardia”
print(Character.birthplace)  # prints “Kingdom of Asgardia”
pickie.birthplace = "Helm’s deep"
Character.birthplace = "Narnia"
print(pickie.birthplace)  # prints “Helm’s deep”
print(Character.birthplace)  # prints “Narnia”
print(woodie.birthplace)  # prints “Narnia”
```

Referencing Attributes

Careful to only reference attributes that you know to exist. If you try to reference an attribute that doesn't exist, Python will throw an AttributeError:

```
woodie = Character("Woodie", 15, 7)
print(woodie.age)
```

```
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    print(woodie.age)
AttributeError: 'Character' object has no attribute 'age'
```

Referencing Attributes

This also happens when we try to access an instance attribute via the class.

```
woodie = Character("Woodie", 15, 7)
print(Character.name)
```

```
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    print(Character.name)
AttributeError: type object 'Character' has no attribute 'na
me'
```

Live Coding! Example 2

- Update the attributes of the characters.

Code available in:
github.com/CSwithJC/CIIC3015/blob/main/lecture_08/example_02.py

Class Methods and Equality

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

Google | GH |

`is` operator checks IDENTITY (Same instance)

Objects that are instances of classes that you define are mutable.

We can use the `is` operator to check if two variables refer to the same instance.

```
woodie = Character("Woodie", 15, 7)
pickie = Character("Pickie", 12, 8)

print(woodie is pickie)  # Prints False
print(woodie is woodie)  # Prints True
```

Object Identity

What does this print?

```
player_1 = Character("Woodie", 15, 7)
player_2 = Character("Woodie", 15, 7)

print(player1 is player2)
```

False

Even though these have the same attributes, they are different instances! `is` operator doesn't compare attributes, it checks if they are the same instance.

`is` vs `==`

`is` operator checks if two variables are the exact same instance (i.e. if the data is actually in the same memory location)

`==` operator checks if two variables have the same values

```
a = [1, 2, 3]
b = [1, 2, 3]
c = a

print(a is b)  # Prints False
print(a == b)  # Prints True
print(a is c)  # Prints True
```

But == operator ALSO tests IDENTITY by default

So if we want to check if two classes have the same values we should use the == operator!

```
player_1 = Character("Woodie", 15, 7)
player_2 = Character("Woodie", 15, 7)

print(player_1 == player_2)
```

If we want == method to test equal attributes:

We have to tell python what it means for two Characters to be equal

```
class Character:
    def __init__(self, char_name, str_score, def_score):
        self.health = 100
        self.name = char_name
        self.strength = str_score
        self.defense = def_score

    def __eq__(self, other):
        return self.name == other.name
```

`__eq__` method

Tells Python what it means for two instances of the same object to be equal

- Always named "`__eq__`"
- Always takes in "`self`" as the first parameter to represent the current instance
- Always takes a second parameter to represent the other instance it is being compared to
- Must return True or False with the result of the equality check

== operator

- Now we can use the == operator!

```
player_1 = Character("Woodie", 15, 7)
player_2 = Character("Woodie", 15, 7)

print(player_1 == player_2)  True
```

== operator

- Now we can use the == operator!

```
player_1 = Character("Woodie", 21, 8)
player_2 = Character("Woodie", 15, 2)

print(player_1 == player_2)  True
```

Being equal means having the same names

`__str__` method

Tells Python what is the string representation of the object of a class

- Always named "`__str__`"
- Always takes in "`self`" as the first parameter to represent the current instance
- Always should return a string.

`__str__` method

```
class Character:
    def __init__(self, name):
        self.name = name
```

```
woodie = Character(name="Woodie")
print(woodie)
# Prints <__main__.Character
object at 0x7ff53f22aa00>
```

```
class CharacterWithStr:
    def __init__(self, name):
        self.name = char_name

    def __str__(self):
        return self.name
```

```
woodie = Character(name="Woodie")
print(woodie)
# Prints Woodie
```

Live Coding! Example 3

- Implement the `__eq__` method and test for equality and `__str__` method for printing.

Code available in:
github.com/CSwithJC/CIIC3015/blob/main/lecture_08/example_03.py

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

Google | GH |

Class Methods

Google | GH |

Damage?

Continuing our character example from last class...how do we implement “take damage”?

Let’s say the damage taken is subtracted from the player’s health. And the damage will be other attacker’s strength minus the defense of the person being attacked

Attack: 15
Defense: 7

Attack: 12
Defense: 8

damage = woodie.attack - pickie.defense = 15 - 8 = 7

Damage?

Reminder: this is what our character class looked like:

```
class Character:
    def __init__(self, char_name, str_score, def_score):
        self.health = 100
        self.name = char_name
        self.strength = str_score
        self.defense = def_score
```

Implement the damage when woodie attacks pickie

```
woodie = Character("Woodie", 15, 7)
pickie = Character("Pickie", 12, 8)
```

Character Attacks

```
woodie = Character("Woodie", 15, 7)
pickie = Character("Pickie", 12, 8)

print(pickie.health)  # Prints 100

# Woodie attacks Pickie
damage = woodie.strength - pickie.defense
pickie.health = pickie.health - damage

print(pickie.health)  # prints 93
```

How do we make this logic reusable?

Attack Function

```
def attack(char1, char2):
  """
  Implements char1 attacking char2. Damage is
  0 is char2's defense > char1's strength.
  """
  damage = char1.strength - char2.defense
  if damage > 0:
    char2.health = char2.health - damage

print(pickie.health)  # Prints 100
attack(woodie, pickie)
print(pickie.health)  # prints 93
```

Attack Function

Our attack function is nice in that we can pass in any two characters.

Cons:

Who is attacking who when we do `attack(woodie, pickie)`

Function is defined separately from our character class, even though it is related so closely to characters.

Can we do better?

Class Methods

Think of a class to be a bundle of

- class attributes (which hold data)
- **class methods (which are functions tied to the class)**

We can defined a function inside of our class, which operates on or with a particular instance of the class! This kind of function is called a class method.

Attack Method

```
class Character:
  def __init__(self, char_name, strs, defs):
    self.health = 100
    self.name = char_name
    self.strength = strs
    self.defense = defs

  def attack(self, char2):
    damage = self.strength - char2.defense
    if damage > 0:
      char2.health = char2.health - damage
```

```
woodie = Character("Woodie", 15, 7)
pickie = Character("Pickie", 12, 8)
print(pickie.health) # Prints 100
woodie.attack(pickie)
print(pickie.health) # Prints 93
```

Google

Instance Methods

Instance methods always take a "`self`" parameter. `self` refers to the instance of the class that is being used to call the function, you don't need to pass it in yourself.

```
class <ClassName>:
  def function_name(self, param1, ...):
    # Use `self` to refer to yourself
    <function body>
```

Advanced Game Character

What properties do they have?

- Health
- Attack Strength
- Defense

Let’s add an “inventory”!

- Weapons List
- Can add or remove
- Can only store max of 4

Advanced Game Character

```
class Character:
  max_inventory_size = 4
  def __init__(self, char_name, str_score, def_score):
    self.health = 100
    self.name = char_name
    self.strength = str_score
    self.defense = def_score
    self.inventory = []

  def introduce(self):
    print("Hi my name is " + self.name)

  def store_item(self, item):
    if len(self.inventory) < self.max_inventory_size:
      self.inventory.append(item)
      print("Inventory:", char1.inventory)
    else:
      print("Too much stuff, can't store another item!")

  def remove_item(self, item):
    if item in self.inventory:
      self.inventory.remove(item)
```

```
char1 = Character("Woodie", 15, 7)
char2 = Character("Pickie", 12, 8)
char1.introduce()
print(char1.inventory)

char1.store_item("apple")
char1.store_item("apple")
char1.store_item("sword")
char1.store_item("shield")
char1.store_item("dagger")
char1.store_item("cloak")

char1.remove_item("apple")
char1.store_item("dagger")
```

Google | GH |

Common Class Methods

You've been using these all along and you didn't even know it...

String methods:

```
school = "UPRM"
print(school.upper())
print(school.lower())
```

`.upper()` and `.lower()` are methods of the String class that return the string as uppercase or lowercase.

Common Class Methods

You've been using these all along and you didn't even know it…

List methods:

```
grades = [92, 73, 100]
print(grades.append(100))
print(grades.pop(0))
```

`.append()`, `.extend()`, `.pop()`, `.remove()` are methods of the List class that mutate the underlying list. All are void except for `.pop()`

Common Class Methods

Is `len()` a class method?

```
grades = [92, 73, 100]
print(len(grades))
```

Nope! It is a standalone function. It isn't called on the instance of a list or string (i.e. `grades.len()`)

Terminology

- A “method” is always inside of a Class and called on a class instance
  - Ex: `my_object.method_name()`
- A “function” is always outside of a Class.
  - Ex: `function_name(object)`

Terminology

- A “class” and a “type” are effectively synonyms in Python.
- Class *Instance* vs *Class Definition*:
  - Think of a string. A “String” is a data type that holds on to a collection of characters, and it has a `.upper()` and `.lower()` method (among others).

- An **instance** of a string holds on to a particular collection of characters.

```
name = "Mr Jake"
school = "Fisk"
semester = "almost done"
```

These are all different *instances* of the same *type* (string)

Live Coding! Example 4

- Implement attack method and inventory field to characters/

Code available in:

github.com/CSwithJC/CIIC3015/blob/main/lecture_08/example_04.py

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

Google | GH |

Inheritance

Inheritance

Imagine that each of our characters have their own “type”

- Some are “Sky” characters that can fly
- Some are “Ocean” characters that can swim
- Some are “Ground” characters that can run

Let’s create classes for each of these characters…

Google | GH |

Inheritance

Sky Character

```
class SkyCharacter:
    def __init__(self, name, strength,
                 defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense

    def action(self):
        print("I can Fly!")
```

Ocean Character

```
class OceanCharacter:
    def __init__(self, name, strength,
                 defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense

    def action(self):
        print("I can Swim!")
```

Ground Character

```
class GroundCharacter:
    def __init__(self, name, strength,
                 defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense

    def action(self):
        print("I can Run!")
```

Great! Now we have classes for each of our characters.

Google | GH |

Inheritance

However, notice how **similar** the three characters are:

```
class SkyCharacter:
    def __init__(self, name, strength,
                 defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense

    def action(self):
        print("I can Fly!")
```

```
class OceanCharacter:
    def __init__(self, name, strength,
                 defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense

    def action(self):
        print("I can Swim!")
```

```
class GroundCharacter:
    def __init__(self, name, strength,
                 defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense

    def action(self):
        print("I can Run!")
```

The only difference is in the action they perform.

Google

Inheritance

- We can use **inheritance** to create the common attributes and methods across all characters in a single class and then have each specific character **inherit** those attributes.
- **Inheritance** is the capability of one class to derive or inherit the properties from another class.
- Let's begin by creating a base **Character** class with all of the attributes:

```
class Character:
    def __init__(self, name, strength, defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense
```

Google

Inheritance

- Now that we have the base **Character** class, we can have our three specific character classes **inherit** from Character:

```
class Character:
    def __init__(self, name, strength, defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense
```

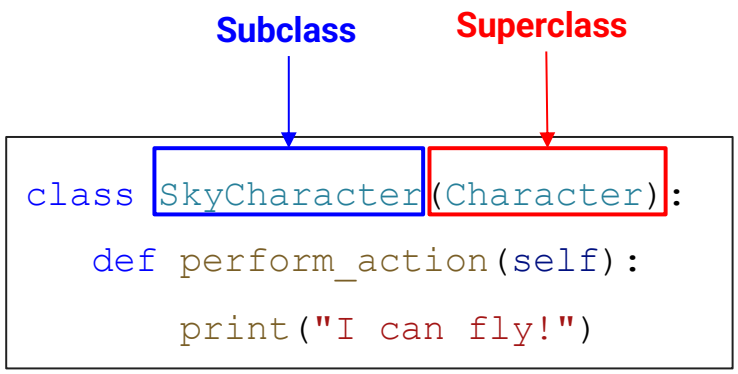
```
class SkyCharacter(Character):
    def perform_action(self):
        print("I can fly!")
```

```
class OceanCharacter(Character):
    def perform_action(self):
        print("I can swim!")
```

```
class GroundCharacter(Character):
    def perform_action(self):
        print("I can run!")
```

Inheritance

Inheritance is denoted in a class by putting the name of the superclass between parentheses after the class name:



- By inheriting from **Character**, **SkyCharacter** now has all of the attributes and methods from **Character** available without having to declare them explicitly.
- Now, **SkyCharacter** can add attributes and methods are are specific to it while also keeping the ones in **Character**.

Google | GH |

Inheritance

- Inheriting from Character means that all of the methods and attributes in Character are now available on **SkyCharacter**, **GroundCharacter** and **OceanCharacter**.

```
class Character:
    def __init__(self, name, strength, defense):
        self.health = 100
        self.name = name
        self.strength = strength
        self.defense = defense
```

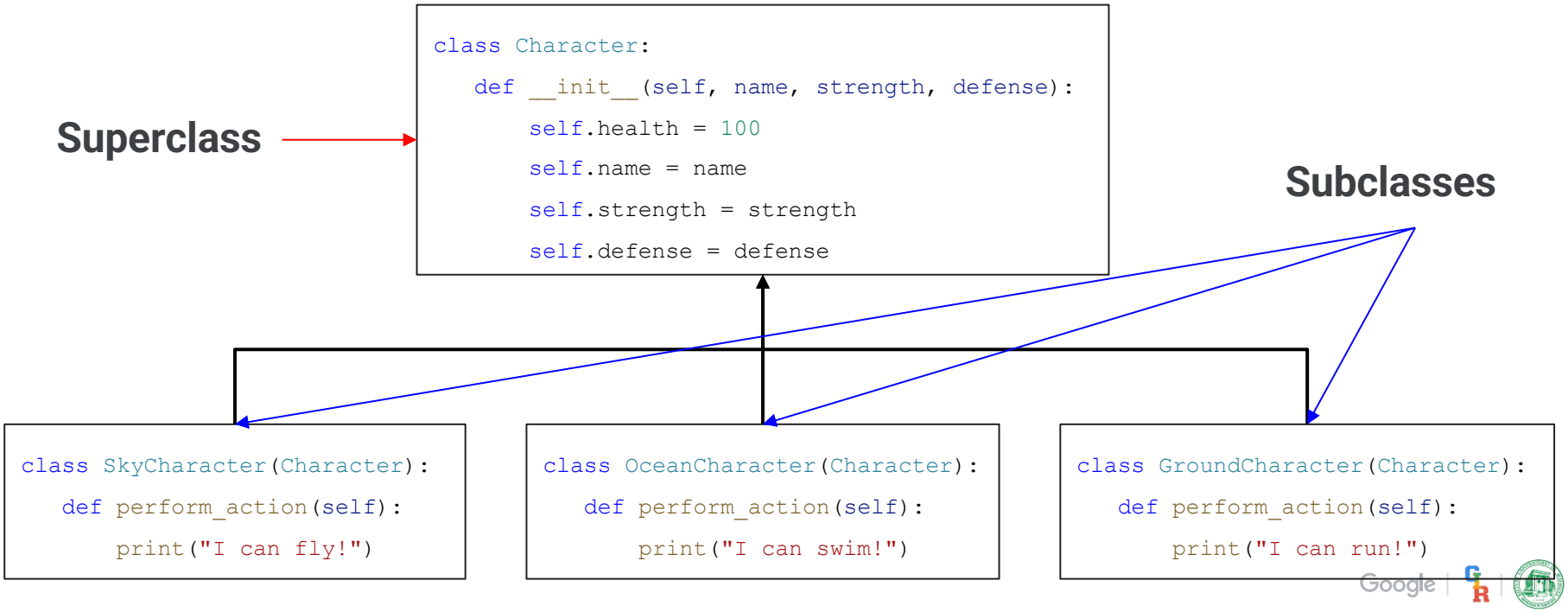
```
class SkyCharacter(Character):
    def perform_action(self):
        print("I can fly!")
```

```
sky = SkyCharacter(name='Sky', strength=15, defense=7)
print(sky.name)  # Prints "Sky"
print(sky.strength)  # Prints 15
print(sky.defense)  # Prints 7
sky.perform_action()  # Prints "I can fly!"
```

Google | GH |

Inheritance

The parent class (Characters) is known as the **superclass**, while all of the classes that inherit from ir are known as **subclasses**.



Inheritance

- We can also **override** (overwrite) inherited methods from superclasses if we want to:

```
class SkyCharacter(Character):
    def __init__(self, name):
        self.name = name

    def perform_action(self):
        print("I can fly!")
```

Overrides the `__init__` from **Character**!

Google | GH |

Inheritance

- However, you would lose all of the attributes and functionality of the method in character

```
class SkyCharacter(Character):
    def __init__(self, name):
        self.name = name

    def perform_action(self):
        print("I can fly!")
```

```
sky = SkyCharacter(name='Sky')
print(sky.name)  # Prints "Sky"
print(sky.strength)  # ERROR!
print(sky.defense)  # ERROR!
```

Inheritance

- What if we want to enhance in the `__init__` of a subclass while keeping the functionality from `__init__` of the superclass?
- We can call the method of the superclass as follows:

```
class SkyCharacter(Character):
    def __init__(self, name, strength, defense):
        Character.__init__(self, name, strength, defense)
        self.sky_points = 1000

    def perform_action(self):
        print("I can fly!")
```

```
sky = SkyCharacter(name='Sky', strength=15, defense=7)
print(sky.name)  # Prints "Sky"
print(sky.strength)  # Prints 15
print(sky.defense)  # Prints 7
print(sky.sky_points)  # Prints 1000
sky.perform_action()  # Prints "I can fly!"
```

- Now, **SkyCharacter** has an enhanced `__init__` method.

Google | GH |

`isinstance()`

- You can use the `isinstance()` function to check if an object is an instance of a specific class.
- For example, let's write a function that prints all attributes of a character, including sky_points if the character is a **SkyCharacter**:

```
sky = SkyCharacter(name='Sky',
strength=15, defense=7)
print_all_attributes(sky)
```

```
def print_all_attributes(character):
    print(character.name)  # Prints "Sky"
    print(character.strength)  # Prints 15
    print(character.defense)  # Prints 7
    if isinstance(character, SkyCharacter):
        print(character.sky_points)  # Prints 1000
    character.perform_action()  # Prints "I can fly!"
```

- This ability of Python (or any programming language) to know what attributes and methods of an object to run based on the class of the object is known as **Polymorphism**.

Live Coding! Example 5

- Implement Inheritance to characters.

Code available in:

github.com/CSwithJC/CIIC3015/blob/main/lecture_08/example_05.py

Live Coding Extra Practice! Example 6

- Implement Student class.

Code available in:

github.com/CSwithJC/CIIC3015/blob/main/lecture_08/example_06.py

Questions?

Post to our Discord server
or
email: upr-ciic3015-staff@upr.edu

Give us anonymous feedback!

bit.ly/CIIC3015_S25_Feedback